Implementation of Decoders for LDPC Block Codes and LDPC Convolutional Codes Based on GPUs

Yue Zhao and Francis C.M. Lau, *Senior Member, IEEE*

I. Introduction

LOW-DENSITY parity-check (LDPC) codes were invented by Robert Gallager [1] but had been ignored for years until Mackay rediscovered them [2]. They have attracted much attention recently because they can achieve excellent error correcting performance based on the belief propagation (BP) decoding algorithm.

However, the BP decoding algorithm requires intensive computations. For applications like optical communication [3], [4] which requires BERs down to 10^{-15}, using CPU-based programs to simulate the LDPC decoder is impractical. Fortunately, the decoding algorithm possesses a high data-parallelism feature, i.e., the data used in the decoding process are manipulated in a very similar manner and can be processed separately from one another. Thus, practical decoders with low-latency and high-throughput can be implemented with dedicated hardware such as field programmable gate arrays (FPGAs) or application specific integrated circuits (ASICs) [5], [6], [7], [8], [9], [10], [11]. However, high performance FPGAs and ASICs are very expensive and are non-affordable by most researchers. Such hardware solutions also cost a long time to develop. In addition, the hardware control and interconnection frame are always associated with a specific LDPC code. If one parameter of an LDPC code/decoder changes, the corresponding hardware design has to be changed accordingly, rendering the hardware-based solutions non-flexible and non-scalable.

Recently, graphics processing units (GPUs) used to process graphics only have been applied to support general purpose computations [12]. In fact, GPUs are highly parallel structures with many processing units. They support floating point arithmetics and can hence conduct computations with the same precision as CPUs. GPUs are particularly efficient in carrying out the same operations to a large amount of (different)

Yue Zhao and Francis Lau are with the Department of Electronic and Information Engineering, The Hong Kong Polytechnic University.

data. Compared with modern CPUs, GPUs can also provide much higher data-parallelism and bandwidth. Consequently, GPUs can provide a cheap, flexible and efficient solution of simulating an LDPC decoder. Potentially, the simulation time can be reduced from months to weeks or days when GPUs, instead of CPUs, are used. In addition, the GPU programming codes can be re-used without much modification should more advanced GPUs be produced by manufacturers.

In [13], [14], a compressed parity-check matrix has been proposed to store the indices of the passing messages in a cyclic or quasi-cyclic LDPC code. Further, the matrix is stored in the constant cache memory on the GPU for fast access. The messages are stored in a compressed manner such that the global memory can be accessed in a coalesced way frequently. However, the coalesced memory access occurs only during the data-read process and is not always guaranteed due to a lack of data alignment. In [12], [15], [16], the sum-product LDPC decoder and the min-sum decoder have been implemented with GPUs. Moreover, by combining sixteen fixed-point 8-bit data to form one 128-bit data, the LDPC decoder in [12] decodes sixteen codewords simultaneously and achieves a high throughput. Although the method in [12] allows coalesced memory access in *either* the read *or* write process, coalesced memory access in *both* the read *and* write processes is yet to be achieved.

Furthermore, the LDPC convolutional codes (LDPCCCs), first proposed in [17], have been shown to achieve a better error performance than the LDPC block code counterpart of similar decoding complexity. There are many features of LDPCCC that make it suitable for real applications. First, the LDPCCC inherits the structure of the convolutional code, which allows continuous encoding and decoding of variable-length codes. Thus the transmission of codewords with varying code length is possible. Second, the LDPCCC adopts a pipelined decoding architecture — in the iterative decoding procedure, each iteration is processed by a separate processor and the procedure can be performed in parallel. So a high-throughput decoder architecture is possible. In [18], [19], the concepts and realization of highly parallelized decoder architectures have been presented and discussed. To the author's best knowledge, there is not any GPU-based implementation of the LDPCCC decoder yet. The reason may lie in the complexity structure of the LDPCCC compared to the LDPC block code, particularly the random time-varying LDPCCC.

As will be discussed in this paper, an LDPCCC derived from a well designed QC-LDPC code possesses not only the good BER performance, but also the regular structure that results in many advantages in practical implementations. Due to the structure inherited from the QC-LDPC code, the LDPCCC decoder enables an efficient and compact memory storage of the messages with a simple address controller.

In this paper, we develop flexible and highly parallel GPU-based decoders for the LDPC codes. We improve the efficiency by making (i) the threads of a warp follow the same execution path (except when deciding whether a bit is a “0” or a “1”) and (ii) the memory accessed by a warp be of a certain size and be aligned. The results show that the decoders based on the GPUs achieve remarkable speed-up improvement — more than 100 times faster than the serial CPU-based decoder.

We also develop a GPU-based decoder for the LDPC convolutional codes. We propose a decoder architecture for LDPCCC derived from QC-LDPC block-code. By taking advantage of the homogeneous operations of the pipeline processors, we compress the index information of different processors into one lookup table. Combined with an efficient thread layout, the decoder is optimized in terms of thread execution and memory access. Simulation results show that compared with the serial CPU-based decoder, the GPU-based one can achieve as many as 200 times speed-up. The GPU-based decoder, moreover, outperforms a quad-core CPU-based decoder by almost 40 times in terms of simulation time.

The rest of the paper is organized as follows. Section II reviews the structure and decoding algorithm of the LDPC code. The same section also reviews the construction of LDPCCC based on QC-LDPC code as well as the decoding process for the LDPCCC. In Section III, the architecture of CUDA GPU and the CUDA programming model is introduced. Section IV describes the implementation of the LDPC decoder and LDPCCC decoder based on GPUs. Section V presents the simulation results of the LDPC decoder and LDPCCC decoder. The decoding times are compared when (i) a GPU is used, (ii) a quad-core CPU is used with a single thread, and (iii) a quad-core CPU is used with up to 8 threads. Finally, Section VI concludes the paper.

II. Review of LDPC Codes and LDPC Convolutional Codes

A. Structure of LDPC Codes and QC-LDPC Codes

A binary (N, K) LDPC code is a linear block code specified by a sparse $M \times N$ parity-check matrix $\mathbf{H}$, where $M = N - K$. The code rate of such an LDPC code is $R \geq K/N = 1 - M/N$. The equality holds when $\mathbf{H}$ is full rank.

The $\mathbf{H}$ matrix contains mostly $0's$ and relatively a small number of $1's$. Such a sparsity structure is the key characteristic that guarantees good performance of LDPC codes. A *regular* LDPC code is a linear block code with $\mathbf{H}$ containing a constant number w_c of 1's in each column and a constant number w_r of 1's in each row. Moreover, w_r and w_c satisfy the equation $w_r = w_c \times \frac{N}{M}$. Otherwise the code is defined as an *irregular* LDPC code.

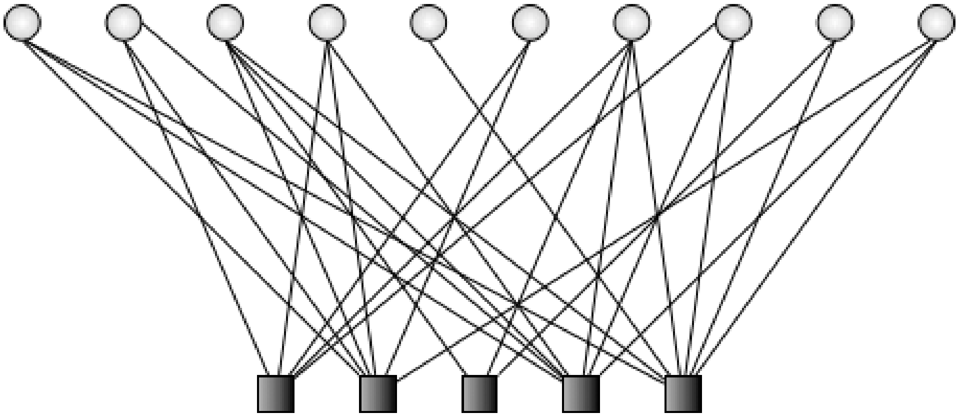


Fig. 1: Tanner Graph representation of the LDPC code defined by (1).

Example 1. *The parity-check matrix* $\mathbf{H}$ *in (1) shows an example of an irregular LDPC code.*

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{1}$$

A bipartite graph called Tanner graph [20] can be used to represent the codes and to visualize the message-passing algorithm. Figure 1 is the underlying Tanner graph of the $\mathbf{H}$ in (1). The N upper nodes are called the message nodes or the variable nodes and the M nodes in the lower part of Fig. 1 are called the check nodes. An edge in the Tanner graph represents the adjacency of the variable node i and the check node j. It corresponds to a nonzero (i, j)-th entry in the $\mathbf{H}$ matrix.

QC-LDPC codes form a subclass of LDPC codes with the parity-check matrix consisting of circulant permutation matrices [21], [22]. The parity-check matrix of a regular (J, L) QC-LDPC code is represented by

$$\mathbf{H} = \begin{bmatrix} \mathbf{P}^{a_{1,1}} & \mathbf{P}^{a_{1,2}} & \cdots & \mathbf{P}^{a_{1,L}} \\ \mathbf{P}^{a_{2,1}} & \mathbf{P}^{a_{2,2}} & \cdots & \mathbf{P}^{a_{2,L}} \\ \vdots & \cdots & \cdots & \vdots \\ \mathbf{P}^{a_{J,1}} & \mathbf{P}^{a_{J,2}} & \cdots & \mathbf{P}^{a_{J,L}} \end{bmatrix}, \tag{2}$$

where J denotes the number of block rows, L is the number of block columns, $\mathbf{P}$ is the identity matrix of size $p \times p$, and $\mathbf{P}^{a_{j,l}}$ $(1 \leq j \leq J;\ 1 \leq l \leq L)$ is a circulant matrix formed by shifting the columns of $\mathbf{P}$ cyclically to the right $a_{j,l}$ times with $a_{j,l}$'s being non-negative integers less than p. The code rate R of $\mathbf{H}$ is lower bounded by $R \geq 1 - J/L$. If one or more of the sub-matrix(matrices) is/are substituted by the zero matrix rendering non-uniform distributions of the check-node degrees or variable-node degrees, the QC-LDPC code becomes an irregular code.

B. Belief Propagation Decoding Algorithm for LPDC Codes

LDPC codes are most commonly decoded using the belief propagation (BP) algorithm [23], [24]. Referring to the Tanner graph shown in Fig. 1, the variable nodes and the check nodes exchange soft messages iteratively based on the connections and according to a two-phase schedule.

Given a binary (*N, K*) LDPC code with a parity-check matrix $\mathbf{H}$, we define $\mathcal{C}$ as the set of binary codewords $\mathbf{c}$ that satisfy the equation $\mathbf{c}\mathbf{H}^{\mathrm{T}} = \mathbf{0}$. At the transmitter side, a binary codeword $\mathbf{c} = (c_0, c_1, \ldots, c_{N-1})$ is mapped into the sequence $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ according to $x_n = 1 - 2c_n$. We assume that $\mathbf{x}$ is then transmitted over an additive white Gaussian noise (AWGN) channel and the received signal vector is then given by $\mathbf{y} = (y_0, y_1, \ldots, y_{N-1}) = \mathbf{x} + \mathbf{g}$, where $\mathbf{g} = (g_0, g_1, \ldots, g_{N-1})$ consists of independent Gaussian random variables with zero mean and variance $\sigma^2 = N_0/2$.

Let μ_n be the initial log-likelihood ratio (LLR) that the variable node n is a “0” to that it is a “1”, i.e.,

$$\mu_n = \ln\left(\frac{\Pr(c_n = 0|y_n)}{\Pr(c_n = 1|y_n)}\right). \tag{3}$$

Initially, μ_n is calculated by $\mu_n = (4/N_0) \cdot y_n = \frac{2y_n}{\sigma^2}$ [25]. Define $\mathcal{N}(m)$ as the set of variable nodes that participate in check node m and $\mathcal{M}(n)$ as the set of check nodes connected to variable node n. At iteration i, let $\beta_{mn}^{(i)}$ be the LLR messages passed from variable node n to check node m; $\alpha_{mn}^{(i)}$ be the LLR messages passed from check node m to variable node n; and $\beta_n^{(i)}$ be the *a posteriori* LLR of variable node n. Then the standard BP algorithm can be described in Algorithm 1 [2], [26].

Note that the decoding algorithm consists of 4 main procedures: initialization, horizontal step, vertical step and making hard decisions. For each of these procedures, multiple threads can be used in executing the computations in parallel and all the threads will follow the same instructions with no divergence occurring, except when making hard decisions.

C. Structure of LDPC Convolutional Codes

A (time-varying) semi-infinite LDPC convolutional code can be represented by its parity check matrix in (3). where m_s is referred to as the syndrome former memory of the parity-check matrix. Besides, the sub-matrices $\mathbf{H}_i(t), i = 0, 1, ..., m_s$, are binary $(c-b) \times c$ matrices given by

$$\mathbf{H}_i(t) = \begin{bmatrix} h_i^{(1,1)}(t) & \cdots & h_i^{(1,c)}(t) \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ h_i^{(c-b,1)}(t) & \cdots & h_i^{(c-b,c)}(t) \end{bmatrix}.$$

If $\mathbf{H}_i(t)$ are full rank for all time instant t, the matrix $\mathbf{H}$ in (3) defines a rate $R = b/c$ convolutional code ignoring the irregularity at the beginning.

$$\mathbf{H}_{[0,\infty]} = \begin{bmatrix} \mathbf{H}_0(0) & & & & & \\ \mathbf{H}_1(1) & \mathbf{H}_0(1) & & & & \\ \vdots & \vdots & \ddots & & & \\ \mathbf{H}_{m_s}(m_s) & \mathbf{H}_{m_s-1}(m_s) & \cdots & \mathbf{H}_0(m_s) & & \\ & \mathbf{H}_{m_s}(m_s+1) & \mathbf{H}_{m_s-1}(m_s+1) & \cdots & \mathbf{H}_0(m_s+1) & \\ & & \ddots & & & \ddots \\ & & \mathbf{H}_{m_s}(t) & \mathbf{H}_{m_s-1}(t) & \cdots & \mathbf{H}_0(t) \\ & & \ddots & \ddots & & \ddots \end{bmatrix}, \qquad (3)$$

Definition 1. *A LDPC convolutional code is called a regular (m_s, J, K)-LDPC convolutional code if the parity-check matrix $\mathbf{H}_{[0,\infty]}$ has exactly K ones in each row and J ones in each column starting from the $(m_s \cdot (c-b)+1)$-th row and $(m_s \cdot c + 1)$-th column.*

Definition 2. *An (m_s, J, K)-LDPC convolutional code is periodic with period T if $\mathbf{H}_i(t), i \in \mathbb{Z}^+$ is periodic, i.e., $\mathbf{H}_i(t) = \mathbf{H}_i(t+T), \forall i, t$.*

A code sequence $\mathbf{v}_{[0,\infty]} = [\mathbf{v}_0, \mathbf{v}_1, ..., \mathbf{v}_\infty]$ is "valid" if it satisfies the equation

$$\mathbf{v}_{[0,\infty]}\mathbf{H}^T_{[0,\infty]} = \mathbf{0} \qquad (8)$$

where $\mathbf{v}_i = (v_i^{(1)}, v_i^{(2)}, ..., v_i^{(c)})$ and $\mathbf{H}^T_{[0,\infty]}$ is the syndrome-former (transposed parity-check) matrix of $\mathbf{H}_{[0,\infty]}$.

D. Deriving LDPC Convolutional codes from QC-LDPC block codes

There are several methods to construct LDPC convolutional codes from LDPC block codes. One method is to derive time-varying LDPCCC by unwrapping randomly constructed LDPC block codes [17] and another is by unwrapping the QC-LDPC codes [27], [28]. We now consider a construction method by unwrapping a class of QC-LDPC block code.

Suppose we have created a (J, L) QC-LDPC block code $\mathbf{H}_{QC}$ with J block-rows and L block-columns. The size of its circulant matrices is $p \times p$. We can derive the parity-check matrix for a LDPC convolutional code using the following steps.

1) Partition the $pJ \times pL$ parity-check matrix $\mathbf{H}_{QC}$ to form a $\Lambda \times \Lambda$ matrix, where Λ is the greatest

common divisor of J and L, i.e.,

$$\mathbf{H}_{QC}=\begin{bmatrix}\mathbf{H}_{1,1} & \mathbf{H}_{1,2} & \cdots & \mathbf{H}_{1,\Lambda}\\ \mathbf{H}_{2,1} & \mathbf{H}_{2,2} & \cdots & \mathbf{H}_{2,\Lambda}\\ \vdots & \vdots & \cdots & \vdots\\ \mathbf{H}_{\Lambda,1} & \mathbf{H}_{\Lambda,2} & \cdots & \mathbf{H}_{\Lambda,\Lambda}\end{bmatrix}_{\Lambda\times\Lambda},$$

where $\mathbf{H}_{i,j}$ is a $(pJ/\Lambda)\times(pL/\Lambda)$ matrix, for $i,j=1,2,...,\Lambda$.

2) Divide $\mathbf{H}_{QC}$ along the diagonal into two portions: the strictly upper-triangular portion $\mathbf{H}_{QC}^{(U)}$ and the lower-triangular portion $\mathbf{H}_{QC}^{(L)}$ as follows:

$$\mathbf{H}_{QC}^{(U)}=\begin{bmatrix}\mathbf{0} & \mathbf{H}_{1,2} & \mathbf{H}_{1,3} & \cdots & \mathbf{H}_{1,\Lambda}\\ & \mathbf{0} & \mathbf{H}_{2,3} & \cdots & \mathbf{H}_{2,\Lambda}\\ & & \ddots & \cdots\vdots & \\ & & & \mathbf{0} & \mathbf{H}_{\Lambda-1,\Lambda}\\ & & & & \mathbf{0}\end{bmatrix}_{\Lambda\times\Lambda},$$

and

$$\mathbf{H}_{QC}^{(L)}=\begin{bmatrix}\mathbf{H}_{1,1} & & & \\ \mathbf{H}_{2,1} & \mathbf{H}_{2,2} & & \\ \vdots & \vdots & \ddots & \\ \mathbf{H}_{\Lambda,1} & \mathbf{H}_{\Lambda,2} & \cdots & \mathbf{H}_{\Lambda,\Lambda}\end{bmatrix}_{\Lambda\times\Lambda}.$$

3) Unwrap the parity-check matrix of the block code to obtain the parity-check matrix of LPDCCC. First paste the strictly upper-triangular portion below the lower-triangular portion. Then repeat the resulting diagonally-shaped matrix infinitely, i.e.,

$$\mathbf{H}_{conv}=\begin{bmatrix}\mathbf{H}_{QC}^{L} & & \\ \mathbf{H}_{QC}^{U} & \mathbf{H}_{QC}^{L} & \\ & \mathbf{H}_{QC}^{U} & \mathbf{H}_{QC}^{L}\\ & & \ddots & \ddots\end{bmatrix}.$$

The resulting time-varying LDPCCC has a period of $T=\Lambda$ and the memory m_s equals $\Lambda-1$. The girth of the derived LPDCCC is at least as large as the girth of the QC-LDPC code [29]. A convenient feature of this time-varying unwrapping is that a family of LDPC convolutional codes can be derived by choosing different circulant size p of the QC-LDPC block code.

Example 2. *Consider a QC-LDPC code with* 4 *block rows and* 24 *block columns, i.e.,* $J=4$ *and* $L=24$*. It is first divided into* 4×4 *equally sized sub-blocks*[1]*, i.e.,* $\Lambda=4$*. Then the parity-check matrix of LDPCCC is derived. The construction process is shown in Fig. 2.*

[1] Here we use sub-block to denote the $(pJ/\Lambda)\times(pL/\Lambda)$ matrix as to distinguish it with the sub-matrix within it, i.e., the $p\times p$ matrix.

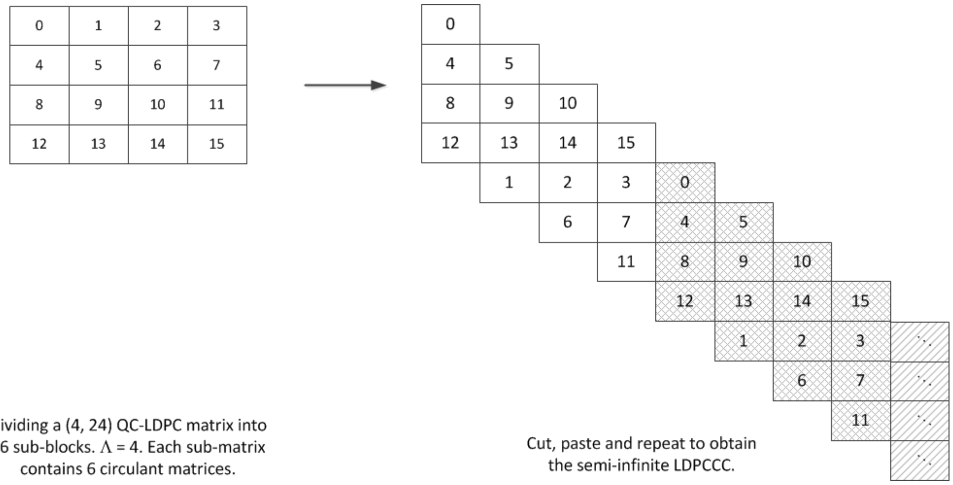

Dividing a (4, 24) QC-LDPC matrix into 16 sub-blocks. Λ = 4. Each sub-matrix contains 6 circulant matrices.
Cut, paste and repeat to obtain the semi-infinite LDPCCC.

Fig. 2: Illustration of constructing a LPDCCC from a QC-LDPC block code.

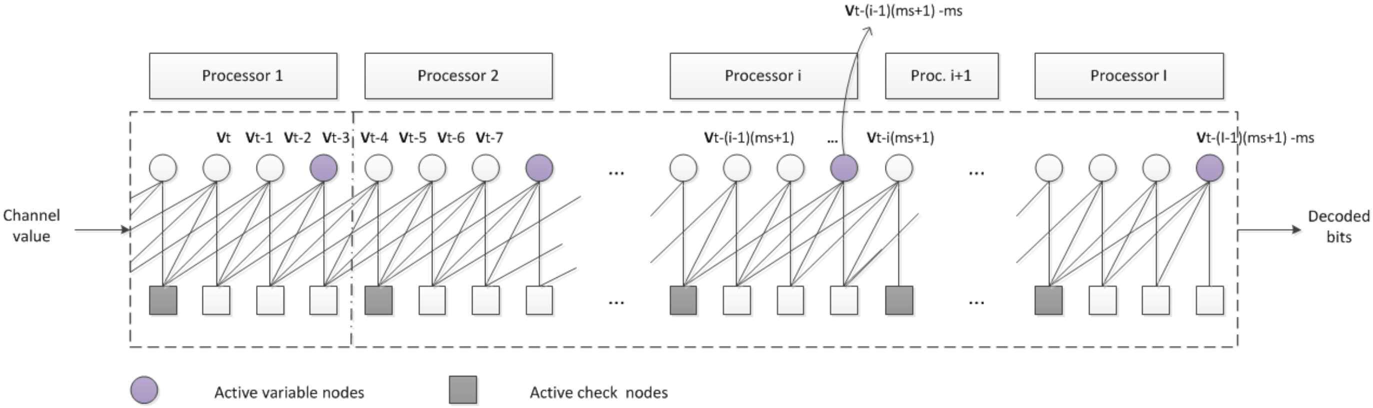


Fig. 3: Continuous decoding of LDPC convolutional code with I processors. Each circle denotes a group of c variable nodes and each square denotes a group of $(c-b)$ check nodes. Each edge represents the connection between the c variable node and the $(c-b)$ check nodes.

E. Decoding Algorithm for LDPCCC

In $\mathbf{H}_{[0,\infty]}$, two different variable nodes connected to the same check node cannot be distant from each other more than m_s time units. This allows a decoding window that operates on a fixed number of nodes at one time. Since any two variable nodes that are at least m_s+1 units apart can be decoded independently, parallel implementation is feasible. The LDPCCC can therefore be decoded with pipelined BP decoding algorithm [17]. Specifically, for a maximum iteration number of I, I independent processors will be employed working on different variable nodes corresponding to different time. In each processor, the variable nodes and the check nodes exchange soft messages iteratively based on the connections and according to a two-phase schedule.

Fig. 3 shows a decoder on the Tanner graph. It is based on the LDPCCC structure shown in Example 2. The code has a rate of $R=5/6$ and a syndrome former memory of $m_s=3$. We refer the c incoming variable nodes (bits) as a frame. Note that every c bits form a frame and every m_s+1 frames are involved in the same constraints. The I processors can operate concurrently. At every iteration, every processor first updates the $(c-b)$ neighboring check nodes of the c variable nodes that just come into this processor. Then every processor will update the c variables which are leaving this processor.

The computations of the check-node updating and variable-node updating are based on the standard BP algorithm Suppose $\mathbf{v}_{[0,\infty]}$ = $[\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_\infty]$, where $\mathbf{v}_t = (v_t^{(1)}, v_t^{(2)}, \ldots, v_t^{(c)})$ is the tth transmitted codeword. Then the codeword $\mathbf{v}_{[0,\infty]}$ is mapped into the sequence $\mathbf{x}_{[0,\infty]} = [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_\infty]$ according to $\mathbf{x}_t = (x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(c)})$ and $x_t^{(j)} = 1 - 2v_t^{(j)}$ $(j = 1, 2, \ldots, c)$. Assuming an AWGN channel, the received signal $\mathbf{y}_{[0,\infty]} = [\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_\infty]$ is further given by $\mathbf{y}_t = (y_t^{(1)}, y_t^{(2)}, \ldots, y_t^{(c)})$ where $y_t^{(j)} = x_t^{(j)} + g_t^{(j)}$ and $g_t^{(j)}$ is an AWGN with zero mean and variance $\sigma^2 = N_0/2$.

Using the same notation as in Sect. II-B, the pipelined BP decoding algorithm applying to LDPCCC is illustrated in Algorithm 2. Same as the LDPC decoding algorithm, the LDPCCC decoding algorithm consists of 4 main procedures: initialization, horizontal step, vertical step and making hard decisions. Moreover, for each of these procedures, multiple threads can be used in executing the computations in parallel and all the threads will follow the same instructions with no divergence occurring, except when making hard decisions.

III. Graphics Processing Unit and CUDA Programming

A graphics processing unit (GPU) consists of multi-threaded, multi-core processors. GPUs follow the single-instruction multiple-data (SIMD) paradigm. That is to say, given a set of data (regarded as a stream), the same operation or function is applied to each element in the stream by different processing units in the GPUs simultaneously. Figure 4 shows a simplified architecture of the latest GPU device. It contains a number of multiprocessors called streaming multiprocessors (or SMs). Each SM contains a group of stream processors or cores and several types of memory including registers, on-chip memory, L2 cache and the most plentiful dynamic random-access memory (DRAM). The L1 cache is dedicated to each multiprocessor and the L2 cache is shared by all multiprocessors. Both caches are used to cache accesses to local or global memory. The on-chip memory has a small capacity (tens of KB) but it has a low latency [30].

In our work, the GPU used is a GTX460, which has 7 SMs and 768 MB global memory. Each SM contains 48 cores [31]. Moreover, the 64 KB on-chip memory is configured as 48 KB shared memory and 16 KB L1 cache for each SM because the more shared memory is utilized, the better.

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by Nvidia. In a CUDA program, computations are performed as a sequence of functions called parallel kernels. Each kernel is typically invoked on a massive number of threads. Threads are first grouped into thread blocks

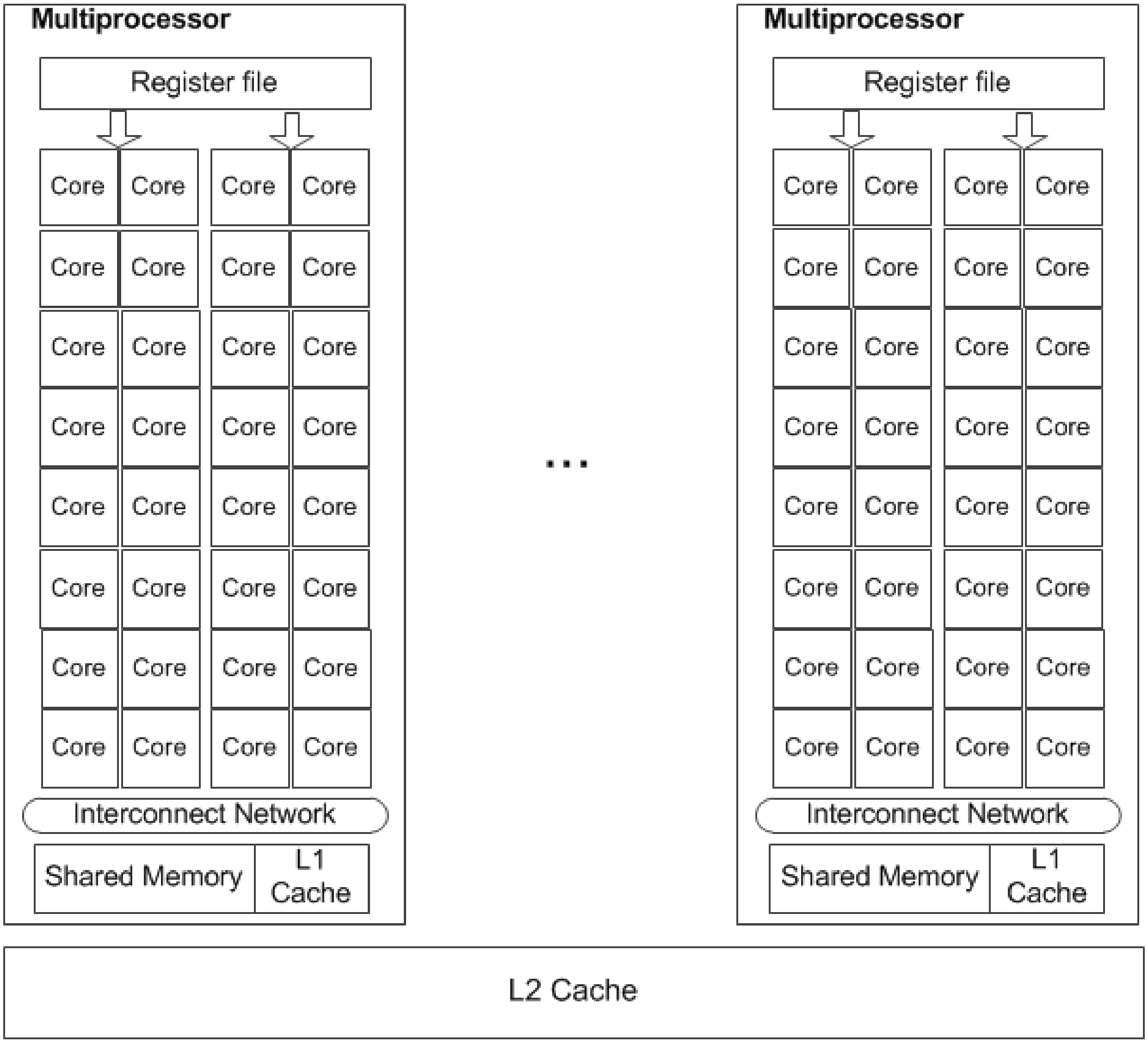


Fig. 4: Hardware architecture of a typical GPU. Each large rectangle denotes a streaming multiprocessor (SM) and each small square in a SM represents a stream processor or core.

and blocks are further grouped into a grid. A thread block contains a set of concurrently executing threads, and the size of all blocks are the same with an upper limit $\Omega_{\max}$. In current GPUs with compute capability 2.x, $\Omega_{\max} = 1024$.

In an abstract level, the CUDA devices use different memory spaces, which have different characteristics. These memory spaces includes global memory, local memory, shared memory, constant memory, texture memory, and registers. The global memory and texture memory are the most plentiful but have the largest access latency followed by constant memory, registers, and shared memory.

CUDA's hierarchy of threads map to a hierarchy of processors on the GPU. A GPU executes one or more kernel grids and a SM executes one or more thread blocks. In current GPUs with compute capability

2.x, the SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. A warp is the execution unit and executes one common instruction at a time. So full efficiency is realized when threads of a warp take the same execution path.

In CUDA programming, the first important consideration is the coalescing global memory accesses. Global memory resides in the device memory and is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned (i.e. the first address is a multiple of their size).

When a warp executes an instruction that accesses the global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads.

IV. Implementation of Decoders for LDPC Codes and LDPCCCs

A. GPU-based LDPC Decoder

We implement our decoders using the standard BP decoding algorithm. According to the CUDA programming model, the granularity of a thread execution and a coalesced memory access is a warp. Full efficiency is realized when all threads in a warp take the same execution path and the coalesced memory access requirement is satisfied. Thus, we propose to decode Γ codewords simultaneously, where Γ is an integer multiple of a warp (i.e., multiple of 32). For each decoding cycle, Γ codewords will be input, decoded, and ouput together and in parallel.

Recall that an LDPC code can be represented by its parity-check matrix or a Tanner graph. A non-zero element in the parity-check matrix corresponds to an edge in the Tanner graph.

In the LDPC decoder, messages are bound to the edges in the Tanner graph (or the 1's in the parity-check matrix $\mathbf{H}$). So we store the messages according to the positions of 1's. Besides, the channel messages corresponding to the variable nodes are required. To reuse the notation, we denote the data structure storing the messages between the variable nodes and the check nodes as $\mathbf{H}$ while the the data structure storing the channel messages as $\mathbf{V}$. The difficulty of the CUDA memory arrangement lies on the fact that for practical LDPC codes with good performance, the positions of the 1's are scattered in the parity-check matrix.

First, in the BP decoding procedure, although there are two kinds of messages, namely, the variable-to-check messages and the check-to-variable messages, at every step of the iteration, only one kind of message is needed to be stored, i.e., after the check-node updating step, only the check-to-variable messages α's are stored in the $\mathbf{H}$ and after the variable-node updating step, only the variable-to-check messages β's are stored in the $\mathbf{H}$. Second, in our new decoder architecture, Γ (Γ is a multiple of a warp) codewords

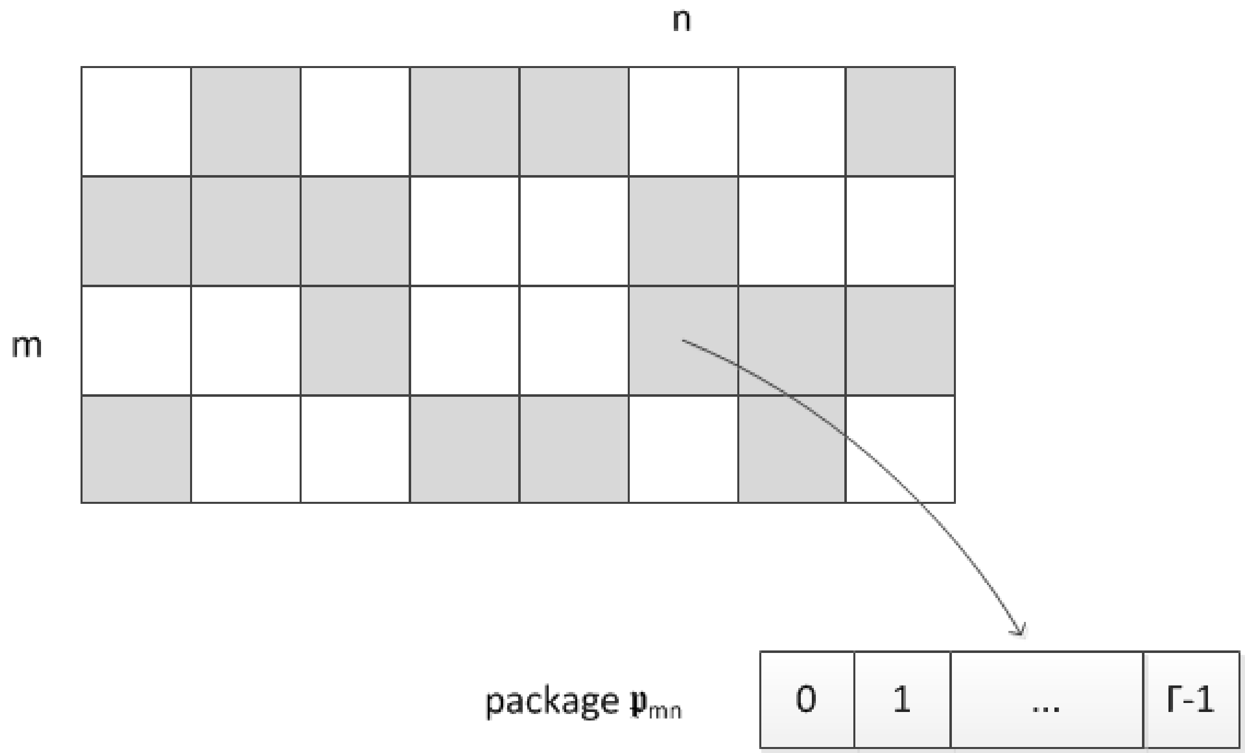


Fig. 5: Data Structure for an LDPC code. Each gray square denotes a non-zero entry in the parity-check matrix. Γ is a multiple of a warp (i.e., multiple of 32).

are decoded together and hence the messages of Γ codewords are also processed together. We number the distinct codewords as $0, 1, ..., \Gamma - 1$ and we use the same notations for the messages as before, i.e., $\beta_{mn}(\gamma)$ is the message from variable node n to check node m corresponding to the γ-th codeword and $\alpha_{mn}(\gamma)$ is the message from check node m to variable node n corresponding to the γ-th codeword. Since all the Γ codewords messages share the same Tanner graph, messages of the Γ distinct codewords corresponding to the same edge can be grouped into one package and stored linearly. Let $\mathfrak{p}_{mn}$ denote the package corresponding to the edge connecting variable node n and check node m. Then in package $\mathfrak{p}_{mn}$, $\beta_{mn}(0), \beta_{mn}(1), ..., \beta_{mn}(\Gamma - 1)$ or $\alpha_{mn}(0), \alpha_{mn}(1), ..., \alpha_{mn}(\Gamma - 1)$ are stored contiguously. This is shown in Figure 5. Different packages $\mathfrak{p}_{mn}$'s are aligned linearly according to their corresponding positions in the parity-check matrix — row-by-row, and left to right for each row. That implies the messages associated to one check node are stored contiguously.

Remark. *To be consistent with the use of memory locations in computer programming, all the indices of the data structures in this paper starts from* 0*.*

The advantage of this arrangement is obvious. Since Γ is a multiple of 32, the memory segment for every package is naturally aligned when the data type belongs to one of the required data types (i.e., with

$$\begin{array}{cc}
\text{variable node } n & \\
\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} & \\
\mathbf{H} = \left[\begin{array}{cccccccc}
0 & 1_{(0)} & 0 & 1_{(1)} & 1_{(2)} & 0 & 0 & 1_{(3)} \\
1_{(4)} & 1_{(5)} & 1_{(6)} & 0 & 0 & 1_{(7)} & 0 & 0 \\
0 & 0 & 1_{(8)} & 0 & 0 & 1_{(9)} & 1_{(10)} & 1_{(11)} \\
1_{(12)} & 0 & 0 & 1_{(13)} & 1_{(14)} & 0 & 1_{(15)} & 0
\end{array}\right] & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \text{check node } m
\end{array} \tag{13}$$

word size of 1-, 2-, 4-, or 8-byte). In addition, the structure of the parity-check matrix $\mathbf{H}$ is shared by the Γ codewords. As these Γ data elements are processed together, they can be accessed by Γ contiguous threads and hence the global memory is always accessed in a coalesced way. We also ensure that the threads within a warp always follow the same execution path with no divergence occurring (except when making hard decisions on the received bits). Then both the memory access and the thread execution are optimal and efficient.

We also need to store the details of the parity-check matrix. Two lookup tables denoted by LUT_c and LUT_v will be kept. LUT_c is used in the check-node updating process and LUT_v is used in the variable-node updating process. The two tables store the indices of the data accessed in the two updating processes and both are two-dimensional. The first dimension is to distinguish different check nodes, i.e., $LUT_c[m]$ is associated with the m-th check node or the m-th row. Each $LUT_c[m]$ records the indices of the messages related to the m-th check node. The two lookup tables are shared by all Γ codewords.

$$LUT_c = \left[\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15
\end{array}\right] \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \text{check node } m \tag{14}$$

$$LUT_v = \left[\begin{array}{cc} 4 & 12 \\ 0 & 5 \\ 6 & 8 \\ 1 & 13 \\ 2 & 14 \\ 7 & 9 \\ 10 & 15 \\ 3 & 11 \end{array}\right] \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \text{variable node } n \tag{15}$$

Example 3. *Consider the parity-check matrix in* (13). *The corresponding data structure begins with the package* $\mathfrak{p}_{01}$, *which is followed by* $\mathfrak{p}_{03}$, $\mathfrak{p}_{04}$, $\mathfrak{p}_{07}$, $\mathfrak{p}_{10}$, $\mathfrak{p}_{11}$, ..., $\mathfrak{p}_{30}$, ..., $\mathfrak{p}_{36}$. *The subscripts of the nonzero entries indicate the sequences (or positions) of the associated data in the entire data structure, starting from* 0. *The* LUT_c *and* LUT_v *are shown in* (14) *and* (15).

It is seen that the size of LUT_c *can be reduced by only storing the first address of the data in each row, namely,* $LUT_c[0]$ *only store* 0, $LUT_c[1]$ *only store* 4 *and so on for all* $m = 0, 1, ..., M-1$. *Particularly, for regular LDPC codes with a unique row weight* d_c, *the indices in* $LUT_c[m]$ *for the* m*-th check node are from* $m \cdot d_c$ *to* $m \cdot d_c + d_c - 1$. *As for the* LUT_v, *the indices are normally irregular and random. Hence a full-indexed lookup table is required for* LUT_v.

The LUT_c *and* LUT_v *lookup tables are stored in the constant or texture memory in the CUDA device so as to be cached to reduce the access time.*

A separate thread is assigned to process each check node or each variable node in the updating kernel. Hence, Γ threads can be assigned to process the data of Γ codewords simultaneously. So, a two dimensional thread hierarchy is launched. The first dimension is for identifying the different codewords while the second dimension is for processing different check nodes or variable nodes. The thread layout is illustrated in Fig. 6. For each thread block, we allocate Γ threads in the threadIdx.x dimension[2], and BL_y threads in the threadIdx.y dimension. Each thread-block contains $BL_y \times \Gamma$ threads, which should be within the thread-block size limit (1024 for the current device). The total number of thread-blocks is determined by the number of check nodes M or the number of variable nodes N. We denote BL_y in the check-node updating kernel as $BL_{y,cnu}$ and the one in the variable-node updating kernel as $BL_{y,vnu}$. Then the numbers of thread blocks are given by $\lceil M/BL_{y,cnu} \rceil$ and $\lceil N/BL_{y,vnu} \rceil$, respectively. In Fig. 6, the threads marked by the vertical rectangular are processing the same codeword.

[2] In CUDA, threads are linear in the threadIdx.x dimension.

In the check-node updating kernel and the variable-node updating kernel, the forward-and-backward calculation is adopted as in [32]. The shared memory is used to cache the involved data so as to avoid re-accessing the global memory. Due to the limited size of the shared memory, the size of the thread-block should not be too large. Consider a (J, L) QC-LDPC code. For each check node, there are $2L$ data elements to be stored. Denote the shared memory size by Z_{shared} and the size of each data by Z_{data}. Consequently in the check-node updating kernel, the thread-block size, denoted by Ω_{cnu}, is limited by

$$\Omega_{\text{cnu}} \leq \frac{Z_{\text{shared}}}{2LZ_{\text{data}}}. \tag{16}$$

In addition, $\Omega_{\text{cnu}} = BL_{y,cnu} \times \Gamma$ and $\Omega_{\text{vnu}} = BL_{y,vnu} \times \Gamma$.

With such a thread layout, the threads access the memory in a straightforward pattern. For example, for the check-node updating kernel, a two-dimensional thread hierarchy with a total size of $\lceil \frac{M}{BL_{y,cnu}} \rceil \times BL_{y,cnu} \times \Gamma$ is launched. During the memory access, every Γ threads are one-to-one mapped to Γ data in a message package. Hence, coalesced memory access is guaranteed.

B. *GPU-based LDPCCC Decoder*

The decoding algorithm and the pipelined LDPCCC decoder architecture have been introduced in Section II-E. The LDPCCCs studied in our work are derived from QC-LDPC codes as described in Section II-D. So our LDPCCC decoder is confined to the LDPCCCs with the parity-check matrix $\mathbf{H}_{[0,\infty]}$ of this kind of structure.

1) Data Structure: The LDPC convolutional codes are decoded continuously. We will thus refer to an LDPCC code sequence $\mathbf{v}_{[0,\infty]}$ = $[\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_\infty]$ as a *code stream* and $\mathbf{v}_i$, $i = 0, 1, \ldots, \infty$ as a *code frame* or *variable frame*. A code stream is constrained with the parity-check matrix $\mathbf{H}_{[0,\infty]}$ by

$$\mathbf{v}_{[0,\infty]}\mathbf{H}^T_{[0,\infty]} = \mathbf{0}.$$

The parity-check matrix of the LDPCCC is shown in Figure 7. It is seen that the check nodes are grouped into layers. Each variable-node frame is connected to $m_s + 1$ (4 here) check layers in the parity-check matrix. Let c denote the size of $\mathbf{v}_i$, $i = 0, 1, ..., \infty$ and $c - b$ denote the size of each check layer. Thus the code rate is b/c.

We will use the same notations as in Section II-D. The LDPCCC is derived from a (J, L) QC-LDPC base code $\mathbf{H}_{QC}$ which has $J \times L$ sub-matrices and the size of each sub-matrix is $p \times p$. $\mathbf{H}_{QC}$ is first divided into $\Lambda \times \Lambda$ sub-blocks[3] ($\Lambda = 4$ in Figure 7) and each sub-block contains several sub-matrices. We

[3] Note that a “sub-block” is different from a “sub-matrix”.

have $c = L/\Lambda \times p$ and $c - b = J/\Lambda \times p$. Referring to Section II-D, we denote the unwrapped parity-check matrix of the QC-LDPC code as

$$\mathbf{H}_{base} = \begin{bmatrix} \mathbf{H}_{QC}^{L} \\ \mathbf{H}_{QC}^{U} \end{bmatrix}.$$

The $\mathbf{H}_{[0,\infty]}$ of the derived LDPCCC is a repetition of $\mathbf{H}_{base}$. Denotingthe number of edges in $\mathbf{H}_{base}$ by E, we have $E = J \times L \times p$.

In designing the LDPCCC decoder, the first thing to consider is the amount of memory required to store the messages. Like the LDPC decoder, we store the messages according to the edges in the parity-check matrix. Let I denote the number of iterations in the LDPCCC decoding. Then I processors are required in the pipelined decoder. Although the parity-check matrix of the LDPCCC is semi-infinite, the decoder only needs to allocate memory for I processors. Hence the total size of the memory required for storing the messages passing between the variable nodes and check nodes is $I \times E$ units. And the total size of the memory required for storing these channel messages is $I \times c$.

Next, we will describe the hierarchical data structure for the LDPCCC decoder memory space. To reuse the notation, we use $\mathbf{H}$ to denote the memory space for the messages on the edges and $\mathbf{V}$ to denote the memory space for the channel messages. The $\mathbf{H}$ is a multi-dimensional array with two hierarchies. First, we divide the entire memory space into I groups corresponding to the I processors and we use the first hierarchy of $\mathbf{H}$ as the data structure for each group. That is $\mathbf{H}[i]$, $i = 0, 1, ..., I-1$ denote the data structure for the I processors, respectively. Second, recall that the parity-check matrix in Figure 7 is derived from $\mathbf{H}_{base}$ which is divided into 16 non-zero sub-blocks and each sub-block has a size of $(pJ/\Lambda) \times (pL/\Lambda)$. Thus in each group, $\mathbf{H}[i]$ is also divided into 16 sub-blocks, denoted by the second hierarchy of $\mathbf{H}$, namely, $\mathbf{H}[i][j]$, where $j = 0, 1, ..., 15$. Every $\mathbf{H}[i][j]$ stores the messages associated with one sub-block. On the other hand, the memory for the channel messages is simpler: $\mathbf{V}[i]$, $i = 0, 1, ..., I \cdot (m_s + 1) - 1$ will be allocated. Finally, to optimize the thread execution and memory access, Γ LDPC convolutional code streams are decoded simultaneously, where Γ is a multiple of a warp. Thus every Γ data are combined into one package and take up one memory unit.

An LDPCCC decoder uses the BP algorithm to update the check nodes and variable nodes. The BP decoding procedures are based on the parity-check matrix $\mathbf{H}_{[0,\infty]}$. With the data structure to store the messages, the decoder also needs the structure information of $\mathbf{H}_{[0,\infty]}$ for understanding the connections between the check nodes and the variable nodes. This information can be used to calculate the index of the data being accessed during the updating. Due to the periodic property of the constructed LDPCCC, the structure of $\mathbf{H}_{base}$ is shared by all the processors. We label the 16 sub-blocks in $\mathbf{H}_{base}$ with the numbers $0, 1, \ldots, 15$.

In addition, in the decoder, the I check-node layers or I variable-node frames being updated simultaneously in the I processors are separated by an interval of m_s+1. Since $\mathbf{H}_{[0,\infty]}$ also has a period of $T=m_s+1$, at any time slot, the I processors require the same structure information in updating the check nodes or the variable nodes, as seen in Figure 7. The lookup tables used in check-node updating and variable-node updating are denoted as LUT_c and LUT_v, respectively. The two lookup tables will then store the labels of the sub-blocks in $\mathbf{H}_{base}$ that are involved in the updating process. Besides, another lookup table LUT_{sub} will be used to store the "shift numbers[4]" of the sub-matrices in each sub-block.

Example 4. *The LUT_c and LUT_v for the LDPCCC in Figure 7 are*

$$LUT_c = \begin{bmatrix} 1 & 2 & 3 & 0 \\ 6 & 7 & 4 & 5 \\ 11 & 8 & 9 & 10 \\ 12 & 13 & 14 & 15 \end{bmatrix} \tag{17}$$

and

$$LUT_v = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 5 & 9 & 13 & 1 \\ 10 & 14 & 2 & 6 \\ 15 & 3 & 7 & 11 \end{bmatrix}. \tag{18}$$

2) Decoding Procedures: Based on the discussion in Section II-E, the detailed decoding procedures are as follows.

1) At time slot 0, the first code frame $\mathbf{v}_0$ enters Processor 1. This means the corresponding memory space $\mathbf{V}[0]$ will be filled with the channel messages of $\mathbf{v}_0$. Then the channel messages will be propagated to the corresponding check nodes. Hence, referring to Fig. 7 and (18), $\mathbf{H}[0][0]$, $\mathbf{H}[0][4]$, $\mathbf{H}[0][8]$ and $\mathbf{H}[0][12]$ will be filled with the same channel messages $\mathbf{V}[0]$.

   Next, the first check layer of $\mathbf{v}_0$, i.e., $\mathbf{c}_0$, will be updated based on the messages from $\mathbf{v}_0$, namely, the messages stored in $\mathbf{H}[0][0]$ (they are the only messages available to $\mathbf{c}_0$).
2) At time slot 1, the second code frame $\mathbf{v}_1$ enters Processor 1. Hence the memory space $\mathbf{V}[1]$, $\mathbf{H}[0][5]$, $\mathbf{H}[0][9]$, $\mathbf{H}[0][13]$ and $\mathbf{H}[0][1]$ will be filled with the messages of $\mathbf{v}_1$. Then, the check layer $\mathbf{c}_1$ are updated in a similar way as the check layer $\mathbf{c}_0$. However, both the messages from $\mathbf{v}_0$ and $\mathbf{v}_1$, i.e., messages stored in $\mathbf{H}[0][4]$ and $\mathbf{H}[0][5]$, are used in the updating of $\mathbf{c}_1$ based on the index information in $LUT_c[1]$. The procedure at time slot 1 is shown in Figure 8a.

[4] For a QC-LDPC base matrix, the information is the "shift number" of each $p\times p$ sub-matrix (-1 represents the all-zero matrix, 0 represents the identity matrix, l represents cyclically right-shifting the identity matrix l times).

The procedure goes on. When $\mathbf{v}_3$ has been input and check layer $\mathbf{c}_3$ has been updated, all the check-to-variable messages needed to update the variable layer $\mathbf{v}_0$ are available. So $\mathbf{v}_0$ will be updated with the channel messages in $\mathbf{V}[0]$ and the check-to-variable messages in $\mathbf{H}[0][0]$, $\mathbf{H}[0][4]$, $\mathbf{H}[0][8]$, and $\mathbf{H}[0][12]$. Now, $\mathbf{v}_0$ is at the end of Processor 1 and is about to be shifted to Processor 2. Instead of copying the memory from one location to another, all we need to do is to specify that the memory $\mathbf{v}_0$ "belongs" to Processor 2.

3) At the next time slot, i.e., time slot 4 (time slot m_s+1), the new code frame $\mathbf{v}_4$ comes. The messages will be stored in $\mathbf{v}[4]$ and $\mathbf{H}[1][0]$, $\mathbf{H}[1][4]$, $\mathbf{H}[1][8]$ and $\mathbf{H}[1][12]$. Now there are two check layers to update, $\mathbf{c}_0$ and $\mathbf{c}_4$. It is noted that $\mathbf{c}_4$ are updated based on all the available messages in $\mathbf{H}[0][1]$, $\mathbf{H}[0][2]$, $\mathbf{H}[0][3]$, and $\mathbf{H}[1][0]$ while $\mathbf{c}_0$ are updated based on the updated messages only in $\mathbf{H}[0][0]$. This insufficient updating of check nodes only occurs to the first m_s code frames. After the updating of the check nodes, the code frame $\mathbf{v}_1$ is at the end of Processor 1 and will be updated. There is no code frame arriving at the end of Processor 2 yet.

4) At time slot $I \cdot (m_s + 1) - 1$, the entire memory space of $\mathbf{V}$ and $\mathbf{H}$ are filled with messages. $\mathbf{v}_0$ and its associated messages are at the end of Processor I (as being labeled) while $\mathbf{v}_{i\cdot(m_s+1)-1}$ is the latest code frame input into Processor 1. Next, the check nodes in the I check layers of the I processors will be updated in parallel. After the updating of the check nodes, all the variable nodes which are leaving Processor i ($i = 1, 2, ..., I$) are updated. Specifically, the variable nodes $\mathbf{v}_{i\cdot(m_s+1)-1}$, $i = 1, 2, \ldots, I$ are to be updated. Furthermore, $\mathbf{v}_0$ is about to leave the decoder. Hard decision will be made based on the *a posteriori* LLR of $\mathbf{v}_0$. Then the memory space of $\mathbf{v}_0$, $\mathbf{V}[0]$, $\mathbf{H}[0][0]$, $\mathbf{H}[0][4]$, $\mathbf{H}[0][8]$ and $\mathbf{H}[0][12]$ are cleared for reuse. At the next time slot $I \cdot (m_s + 1)$, the new code frame $\mathbf{v}_{I\cdot(m_s+1)}$ comes in the decoder and these memory space will be filled with the messages of $\mathbf{v}_{I\cdot(m_s+1)}$.

Remark. *In our GPU-based decoder, all the check nodes (variable nodes) needed to be updated in the I processors are processed in parallel by multiple threads.*

5) Note that the LDPCCC matrix has a period of $T = m_s+1$ (4 here). Hence, at time slot $t \geq I\cdot(m_s+1)$, $\mathbf{v}_t$ enters the decoder and reuses the memory space of $\mathbf{v}_\tau$ where $\tau = t \mod (I \cdot (m_s + 1))$. Furthermore, we let $\kappa = t \mod (m_s + 1)$. Then the check layer $\mathbf{c}_{\kappa+(i-1)\cdot(m_s+1)}$ in Processor i ($i = 1, 2, \ldots, I$) will be updated followed by the updating of the code frame $\mathbf{v}_{\kappa+i\cdot(m_s+1)-m_s}$. Moreover, the "oldest" code frame residing in the decoder — $\mathbf{v}_{t-I\cdot(m_s+1)}$ — is about to leave the decoder and hard decisions will be made on it.

So the entire LDPCCC decoder possesses a circulant structure, as shown in Figure 9. The memory is not shifted except for the one associated with the code frame which is leaving the decoder. Instead, the processor are “moving” by changing the processor label of each code frame. Correspondingly, the “entrance” and “exit” are moving along the data structure. This circulant structure reduces the time for memory manipulation and simplifies the decoder.

3) Parallel Thread Hierarchy: As described in Sect. IV-B1, the memory associated with each entry in the $\mathbf{H}$ matrix is a message package containing Γ messages from Γ code streams. So there is a straightforward mapping between the thread hierarchy and the data structure. In the check-node-updating kernel (or variable-updating-kernel), a two dimensional thread hierarchy of size $I\cdot(c-b)\times\Gamma$ (or $I \cdot c \times \Gamma$) is launched, where $(c - b)$ (or c) is mapped to the total number of check nodes (or variable nodes) being updated in I processors. The size of one of the dimensions (i.e., Γ) is mapped to the number of code streams. Like in LDPC decoder, Γ will be configured as the *threadIdx.x* dimension and $(c - b)$ (or c) will be the *threadIdx.y* dimension in the CUDA thread hierarchy. The Γ threads in the *threadIdx.x* dimension is contiguous and will access the Γ data in each message package for coalesced access.

C. CPU-based LDPC and LDPCCC Decoders

We implement both the serial CPU-based LDPC decoder and LDPCCC decoder using the C language. As CPUs with multiple cores are very common nowadays, we further implement a multi-thread CPU-based LDPCCC decoder using OpenMP. OpenMP [33] is a portable, scalable programming interface for shared-memory parallel computers. It can be used to explicitly direct multi-threaded, shared memory parallelism. A straightforward application of the OpenMP is to parallize the intensive loop-based code with the *#pragma omp parallel for* directive. Then the executing threads will be automatically allocated to different cores on a multi-core CPU.

The horizontal step and the vertical step in Algorithm 2 involve intensive computing. On a single-core CPU, the updating of the different nodes are processed with a serial *for* loop. Since the updating of different nodes can be performed independent of one another, it is ideal to parallelize the *for* loop with the *#pragma omp parallel for* directive in the OpenMP execution on a multicore CPU. Hence, in our implementation, we issue multiple threads to both the updating of the check nodes (9) and the updating of the variable nodes (10) in the multi-thread CPU-based LDPCCC decoder.

V. Results and Discussion

A. The Experimental Environment

The CPU being used is an Intel Xeon containing 4 cores. Moreover, it can handle up to 8 threads at a time. The serial CPU-based decoders are developed using C and the multi-threaded CPU-based LDPCCC

	CPU	**GPU**
Platform	Intel Xeon	Nvidia GTX460
Number of cores	4	7 × 48 = 336
Clock rate	2.26 GHz	0.81 GHz
Memory	8 GB DDR3 RAM	768 MB global memory and 48 KB shared memory
Maximum number of threads	8	—
Maximum thread-block size	—	1024 threads
Programming language	C/OpenMP	CUDA C

TABLE I: Simulation environments.

Code	$J \times L$	p	$c \times (c - b)$	**Number of Edges**
A	4×24	422	2532×422	40512
B	4×24	632	3792×632	60672
C	4×24	768	4608×768	73728
D	4×24	1024	6144×1024	98304

TABLE II: Parity-check matrices of the QC-LDPC codes used in the LDPC decoder. They are also used to derive the LDPCCCs A' to D'.

decoder is developed using OpenMP. Note that for the serial CPU-based decoders, only one of the 4 cores in the CPU will be utilized. The GPU used in this paper is a GTX460 containing 336 cores and the GPU-based decoders are developed using CUDA C. Furthermore, in our simulations, 32 codewords are decoded simultaneously in the GPU decoders, i.e., $\Gamma = 32$. Details of the CPU and GPU used in our simulations are presented in Table I.

Table II shows the characteristics of the QC-LDPC codes under test. For Code A to code D, $J = 4$ and $L = 24$ thus giving the same code rate of $(24-4)/24 = 5/6$. These codes are further used to derived regular LDPCCCs. In order to avoid confusion, we denote the derived LDPCCCs as Code A' to Code D'. It can be readily shown that the $(3, 4, 24)$-LDPCCCs A' to D' have the same code rate of $5/6$.

Remark. *Note that although QC-LDPC codes are adopted in the simulation, the new GPU-based LDPC decoder is able to decode other LDPC codes like randomly-constructed regular or irregular codes.*

Code	C_{GPU}	T_{GPU} (s)	t_{GPU} (ms)	C_{CPU}	T_{CPU} (s)	t_{CPU} (ms)	Speedup ($\frac{t_{\text{CPU}}}{t_{\text{GPU}}}$)
A	2832	6	2.12	4058	1270	313	148
B	12768	37	2.9	11664	5350	458	158
C	21664	74	3.4	20046	10950	546	161
D	82624	371	4.5	70843	51580	728	162

TABLE III: Decoding time for the GPU-based LDPC decoder and the serial CPU-based decoder at E_b/N_0=3.2 dB. 30 iterations are used. C represents the total number of decoded codewords; T denotes the total simulation time and t is the average simulation time per codeword.

B. The Decoding Time Comparison

In order to optimize the speed and to minimize the data transfer between the CPU (host) and the GPU (device), we generate and process the data, including the codeword and the AWGN noise, directly on the GPU. After hard decisions have been made on the received bits, the decoded bits are transferred to the CPU which counts the number of error bits. Since the data transfer occurs only at the end of the iterative decoding process, the transfer time (overhead) is almost negligible compared with time spent in the whole decoding process.

In the following, we fix the number of decoding iterations and the simulation terminates after 100 block/frame errors are received. By recording the total number of blocks/frames decoded and the total time taken[5], we can compute the average time taken to decode one block/frame.

1) LDPC decoders: The GPU-based decoder and the serial CPU-based decoder are tested with 30 iterations at a E_b/N_0 of 3.2 dB. Table III shows the number of transmitted codewords and the simulation times for different codes.

We consider the average time for decoding one codeword for the serial CPU-based decoder, i.e., t_{CPU}. We observe that t_{CPU} increases from Code A to Code D due to an increasing number of edges in the codeword. Further, we consider the average time for decoding one codeword for the GPU-based decoder, i.e., t_{GPU}. Similar to the serial CPU-based decoder, t_{GPU} increases from Code A to Code D.

Finally, we compare the simulation times of the serial CPU-based decoder and the GPU-based decoders by taking the ratio $t_{\text{CPU}}/t_{\text{GPU}}$. The results in Table III indicate that the GPU-based decoder accomplishes speedup improvements from 148 times to 162 times compared with the serial CPU-based decoder.

[5] In the case of the GPU-based decoders, the total time taken includes the GPU computation time, the time spent in transferring data between the CPU and GPU, etc. However, as explained above, the GPU computation time dominates the total time while the overhead is very small.

Code	Number of threads used				
	1	2	4	6	8
A’	39	20	11	10	9
C’	73	38	21	19	17

TABLE IV: Average decoding time (ms) per code frame for the quad-core CPU-based decoder when different numbers of threads are used.

2) LDPCCC decoders: We decode the LDPC convolutional codes A’ to D’ at a E_b/N_0 of 3.1 dB with $I = 20$. First, we show the average decoding times for Code A’ and Code C’ when different numbers of threads are used in the CPU-based decoders. The results are shown in Table IV. The serial CPU-based decoder corresponds to the case with a single thread. We observe that the decoding time is approximately inversely proportional to the number of threads used — up to 4 threads. However, the time does not improve much when the number of threads increases to 6 or 8. The reason is as follows. The CPU being used has 4 cores, which can execute up to 4 tasks in fully parallel. Hence, compared with using a single thread, there is an almost 4 times improvement when 4 threads are used. As the number of threads increases beyond 4, however, all the threads cannot really be executed at the same time by the 4 cores. Consequently, further time improvement is small when more than 4 threads are used.

Next, we compare the decoding times of the LDPCCC decoders when GPU-based and CPU-based decoders are used to decode Code A’ to Code D’. For the CPU-based decoders, we consider the cases where a single thread and 8 threads are used, respectively. Table V shows the results. As explained above, limited by the number of cores (4 only) in the CPU, the CPU-based decoder can only improve the speed by about 4 times even when the number of threads increases from 1 to 8. We also observe that compared with the serial CPU-based decoder, the GPU-based LDPCCC decoder can achieve 170 to 200 times speedup improvement. Compared with the 8-thread CPU-based decoder, the GPU-based LDPCCC decoder can also accomplish 39 to 46 times speedup improvement.

VI. Conclusion

In this paper, efficient decoders for LDPC codes and LDPC convolutional codes based on the GPU parallel architecture are implemented. By using efficient data structure and thread layout, the thread divergence is minimized and the memory can be accessed in a coalesced way. All decoders are flexible and scalable. First, they can decode different codes by changing the parameters. Hence, the programs need very little modification. Second, they should be to run on the latest or even future generations of GPUs which possess more hardware resources. For example, if there are more cores/memory in the GPU,

Code	C_{GPU}	T_{GPU} (s)	t_{GPU} (ms)	C_{CPU}	$T_{\text{CPU}-1}$ (s)	$t_{\text{CPU}-1}$ (ms)	$T_{\text{CPU}-8}$ (s)	$t_{\text{CPU}-8}$ (ms)	$\frac{t_{\text{CPU}-1}}{t_{\text{CPU}-8}}$
A	3136	0.73	0.23	2846	112	39	28	9	4.3
B	6272	1.95	0.31	5716	345	60	79	14	4.3
C	14400	5.4	0.38	13303	976	73	230	17	4.3
D	43680	21.0	0.48	37451	3590	96	834	22	4.4

TABLE V: Decoding time for the GPU-based LDPCCC decoder and the CPU-based decoders at E_b/N_0=3.1 dB. $I = 20$ processors are used. C represents the total number of decoded frames; T denotes the total simulation time and t is the average simulation time per frame. CPU-1 and CPU-8 denote the use of 1 thread and 8 threads, respectively, in the CPU-based decoder.

we can readily decode more codes, say $\Gamma = 64$ codes as compared with $\Gamma = 32$ codes used in this paper, at the same time. These are actually advantages of GPU parallel architecture compared to other parallel solutions including FPGA or VLSI. We will report our results in the future when we have the opportunity to run our proposed mechanism in other GPU families.

Compared with the traditional serial CPU-based decoders, results show that the proposed GPU-based decoders can achieve $100\times$ to $200\times$ speedup. The actual time depends on the particular codes being simulated. When compared with the 8-thread CPU-based decoder, the GPU-based decoder can also accomplish 39 to 46 times speedup improvement. Thus the simulation time can be reduced from months to weeks or days when a GPU-based decoder is used. In summary, our results show that the proposed GPU-based LDPC/LDPCCC decoder has obvious advantages in the decoding time compared with CPU-based decoders.

REFERENCES

[1] R. G. Gallager, *Low-Density Parity-Check Codes*. The MIT Press, Sep. 1963.

[2] D. MacKay, "Good error-correcting codes based on very sparse matrices," *Information Theory, IEEE Transactions on*, vol. 45, no. 2, pp. 399–431, 1999.

[3] I. Djordjevic, M. Cvijetic, L. Xu, and T. Wang, "Using LDPC-Coded modulation and coherent detection for ultra highspeed optical transmission," *Lightwave Technology, Journal of*, vol. 25, no. 11, pp. 3619–3625, 2007.

[4] Y. Miyata, K. Sugihara, W. Matsumoto, K. Onohara, T. Sugihara, K. Kubo, H. Yoshida, and T. Mizuochi, "A triple-concatenated FEC using soft-decision decoding for 100 Gb/s optical transmission," in *Optical Fiber Communication (OFC), collocated National Fiber Optic Engineers Conference, 2010 Conference on (OFC/NFOEC)*, 2010, pp. 1–3.

[5] Y. Chen and D. Hocevar, "A FPGA and ASIC implementation of rate 1/2, 8088-b irregular low density parity check decoder," in *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, vol. 1, 2003, pp. 113–117 Vol.1.

[6] I. B. Djordjevic, M. Arabaci, and L. L. Minkov, "Next generation FEC for High-Capacity communication in optical transport networks," *Journal of Lightwave Technology*, vol. 27, no. 16, pp. 3518–3530, 2009.

[7] B. Levine, R. R. Taylor, and H. Schmit, “Implementation of near Shannon limit error-correcting codes using reconfigurable hardware,” 2000.

[8] A. Pusane, A. Feltstrom, A. Sridharan, M. Lentmaier, K. Zigangirov, and D. Costello, “Implementation aspects of LDPC convolutional codes,” *Communications, IEEE Transactions on*, vol. 56, no. 7, pp. 1060–1069, 2008.

[9] S. Bates, Z. Chen, L. Gunthorpe, A. Pusane, K. Zigangirov, and D. Costello, “A low-cost serial decoder architecture for low-density parity-check convolutional codes,” *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 55, no. 7, pp. 1967 –1976, Aug. 2008.

[10] Z. Chen, S. Bates, and W. Krzymien, “High throughput parallel decoder design for LDPC convolutional codes,” in *Circuits and Systems for Communications, 2008. ICCSC 2008. 4th IEEE International Conference on*, May 2008, pp. 35 –39.

[11] R. Swamy, S. Bates, and T. Brandon, “Architectures for asic implementations of low-density parity-check convolutional encoders and decoders,” in *Proc. IEEE Int. Symp. Circuits and Systems ISCAS 2005*, 2005, pp. 4513–4516.

[12] G. Falcao, V. Silva, and L. Sousa, “How GPUs can outperform ASICs for fast LDPC decoding,” in *Proceedings of the 23rd international conference on Supercomputing*. Yorktown Heights, NY, USA: ACM, 2009, pp. 390–399.

[13] H. Ji, J. Cho, and W. Sung, “Massively parallel implementation of cyclic LDPC codes on a general purpose graphics processing unit,” in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*. IEEE, 2009, pp. 285–290.

[14] ——, “Memory access optimized implementation of cyclic and quasi-cyclic LDPC codes on a GPGPU,” *Journal of Signal Processing Systems*, pp. 1–11, 2010.

[15] G. Falcao, L. Sousa, and V. Silva, “Massive parallel LDPC decoding on GPU,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. Salt Lake City, UT, USA: ACM, 2008, pp. 83–90.

[16] ——, “Massively LDPC decoding on multicore architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 309–322, Feb. 2011.

[17] A. J. Felstrom and K. Zigangirov, “Time-varying periodic convolutional codes with low-density parity-check matrix,” *Information Theory, IEEE Transactions on*, vol. 45, no. 6, pp. 2181–2191, 1999.

[18] M. Tavares, E. Matus, S. Kunze, and G. Fettweis, “A dual-core programmable decoder for LDPC convolutional codes,” in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, May 2008, pp. 532 –535.

[19] E. Matus, M. Tavares, M. Bimberg, and G. Fettweis, “Towards a GBit/s programmable decoder for LDPC convolutional codes,” in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, May 2007, pp. 1657 –1660.

[20] R. Tanner, “A recursive approach to low complexity codes,” *Information Theory, IEEE Transactions on*, vol. 27, no. 5, pp. 533–547, 1981.

[21] M. Fossorier, “Quasicyclic low-density parity-check codes from circulant permutation matrices,” *Information Theory, IEEE Transactions on*, vol. 50, no. 8, pp. 1788–1793, 2004.

[22] W. Tam, F. Lau, and C. Tse, “A class of QC-LDPC codes with low encoding complexity and good error performance,” *Communications Letters, IEEE*, vol. 14, no. 2, pp. 169–171, 2010.

[23] T. Richardson, M. Shokrollahi, and R. Urbanke, “Design of capacity-approaching irregular low-density parity-check codes,” *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 619–637, 2001.

[24] T. Richardson and R. Urbanke, “Efficient encoding of low-density parity-check codes,” *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 638–656, 2001.

[25] X. Hu, E. Eleftheriou, D. Arnold, and A. Dholakia, “Efficient implementations of the sum-product algorithm for decoding LDPC codes,” in *Global Telecommunications Conference, 2001. GLOBECOM ’01. IEEE*, vol. 2, 2001, pp. 1036–1036E vol.2.

[26] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X. Hu, “Reduced-Complexity decoding of LDPC codes,” *Communications, IEEE Transactions on*, vol. 53, no. 8, pp. 1288–1299, 2005.

[27] R. Tanner, D. Sridhara, A. Sridharan, T. Fuja, and D. Costello, “LDPC block and convolutional codes based on circulant matrices,” *Information Theory, IEEE Transactions on*, vol. 50, no. 12, pp. 2966–2984, 2004.

[28] A. E. Pusane, R. Smarandache, P. O. Vontobel, and D. J. Costello, “On deriving good LDPC convolutional codes from QC-LDPC block codes,” in *Proc. IEEE Int. Symp. Information Theory ISIT 2007*, 2007, pp. 1221–1225.

[29] M. Lentmaier, D. G. M. Mitchell, G. P. Fettweis, and D. J. Costello, “Asymptotically regular LDPC codes with linear distance growth and thresholds close to capacity,” in *Proc. Information Theory and Applications Workshop (ITA)*, 2010, pp. 1–8.

[30] C. Nvidia, “Compute Unified Device Architecture Programming Guide Version 4.0,” NVIDIA Corporation, Tech. Rep., 2011.

[31] W. Nvidia, N. Generation, and C. Compute, “Whitepaper nvidia’s next generation cuda compute architecture,” *ReVision*, pp. 1–22, 2009.

[32] F. Kschischang, B. Frey, and H. Loeliger, “Factor graphs and the sum-product algorithm,” *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 498–519, 2001.

[33] R. Chandra, *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

Algorithm 1 BP Decoding Algorithm for LDPC Code

```
[Initialization]
for $0 \leq n \leq N-1$ and $m \in \mathcal{M}(n)$ do
    $\beta_{mn}^{(0)} = \mu_n$
end for
Reset the iteration counter $i = 1$
[Horizontal step]
for $0 \leq m \leq M-1$ and $n \in \mathcal{N}(m)$ do
    Update the check-to-variable messages by
        $$\alpha_{mn}^{(i)} = 2\tanh^{-1}\left(\prod_{n' \in \mathcal{N}(m)\backslash n} \tanh\left(\frac{\beta_{mn'}^{(i-1)}}{2}\right)\right) \tag{4}$$
    where $\mathcal{N}(m)\backslash n$ denotes the set $\mathcal{N}(m)$ excluding the variable node $n$
end for
[Vertical step]
for $0 \leq n \leq N-1$ and $m \in \mathcal{M}(n)$ do
    Update the variable-to-check messages by
        $$\beta_{mn}^{(i)} = \mu_n + \sum_{m' \in \mathcal{M}(n)\backslash m} \alpha_{m'n}^{(i)} \tag{5}$$
    where $\mathcal{M}(n)\backslash m$ denotes the set $\mathcal{M}(n)$ with check node $m$ excluded.
    Calculate the *a posteriori* LLRs using
        $$\beta_{n}^{(i)} = \mu_n + \sum_{m' \in \mathcal{M}(n)} \alpha_{m'n}^{(i)} \tag{6}$$
end for
[Next iteration]
if $i < I_{max}$ (the max. no. of iterations) then
    $i = i + 1$
    Go to the [Horizontal Step]
end if
[Making hard decision]
Make the hard decisions based on the LLRs
        $$\hat{c}_n^{(i)} = \begin{cases} 0 & \text{if } \beta_n^{(i)} \geq 0 \\ 1 & \text{if } \beta_n^{(i)} < 0 \end{cases} \tag{7}$$
```

Algorithm 2 BP Decoding Algorithm for LDPCCC

```
Set time $t = 1$
[Initialization]
Shift $c$ new variable nodes (denoted by $\mathbf{v}_t$) together with their channel messages $\mu_n$ into the first processor.
for $i = 1, 2, ..., I-1$ do
    if $t \geq i(m_s+1)$ then
        Shift the variables $\mathbf{v}_{t-i(m_s+1)}$ along with their associated variable-to-check messages $\beta$'s from the $i$-th processor to the $(i+1)$-th processor.
    end if
end for
for Processor $i$, $i = 1, 2, ..., I$ do
    [Horizontal step]
    Update the $(c-b)$ check nodes corresponding to the $t-(i-1)(m_s+1)$-th block row of $\mathbf{H}_{[0,\infty]}$ (as in (3)) using
        $$\alpha_{mn} = 2\tanh^{-1}\left(\prod_{n' \in \mathcal{N}(m)\backslash n} \tanh\left(\frac{\beta_{mn'}}{2}\right)\right) \tag{9}$$
    [Vertical step]
    Update the variable nodes $\mathbf{v}_{t-(i-1)(m_s+1)-m_s}$ using
        $$\beta_{mn} = \mu_n + \sum_{m' \in \mathcal{M}(n)\backslash m} \alpha_{m'n} \tag{10}$$
end for
[Making hard decision for the variable nodes leaving the last processor]
Evaluate the *a posteriori* LLRs of the frame $\mathbf{v}_{t-(I-1)(m_s+1)-m_s}$ using
    $$\beta_n = \mu_n + \sum_{m' \in \mathcal{M}(n)} \alpha_{m'n} \tag{11}$$
Make hard decisions based on the LLRs
    $$\hat{v}_n = \begin{cases} 0 & \text{if } \beta_n \geq 0 \\ 1 & \text{if } \beta_n < 0 \end{cases} \tag{12}$$
Set time $t = t+1$ and Go to [Initialization]
```

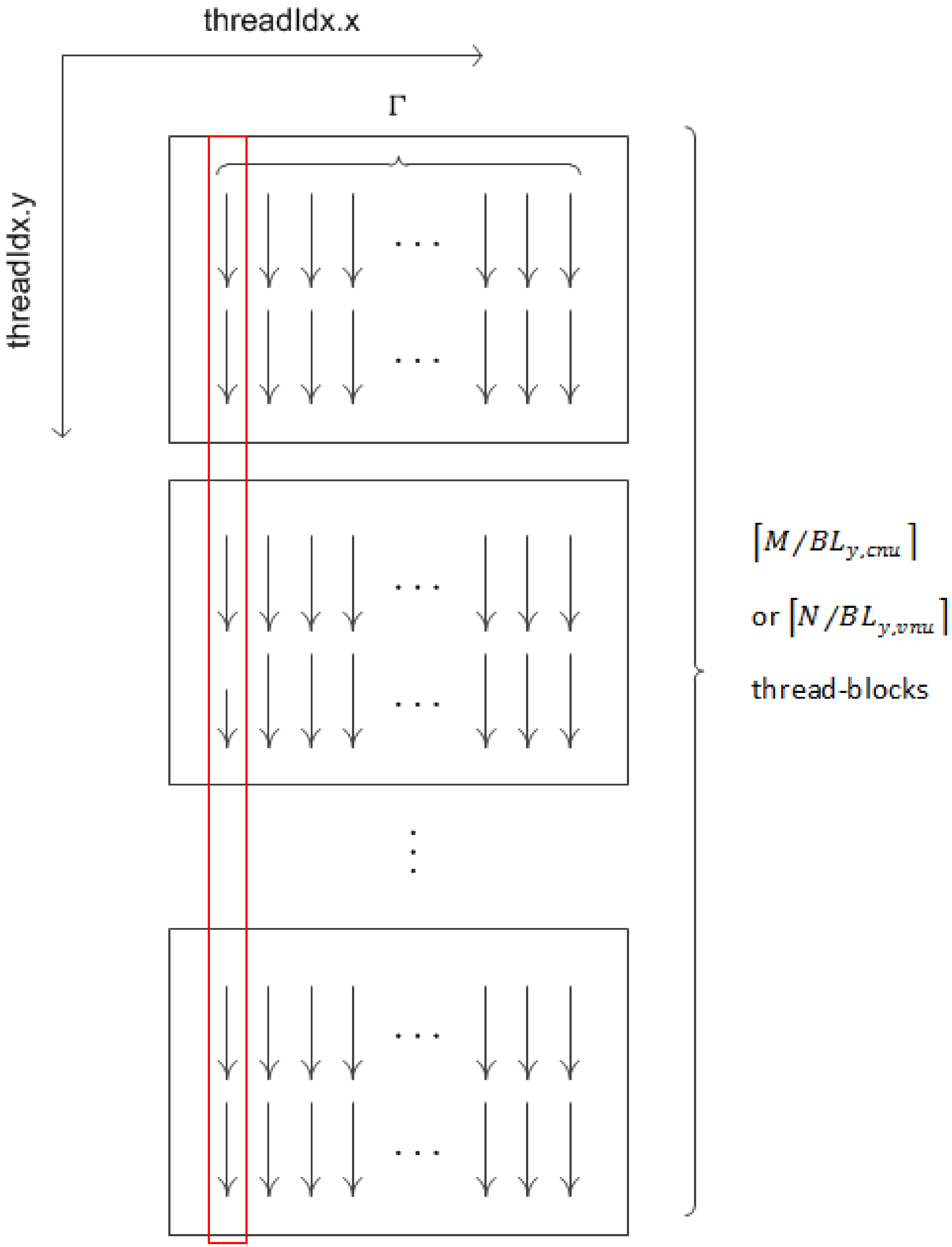


Fig. 6: Two dimensional thread layout of the check-node/variable-node updating kernel.

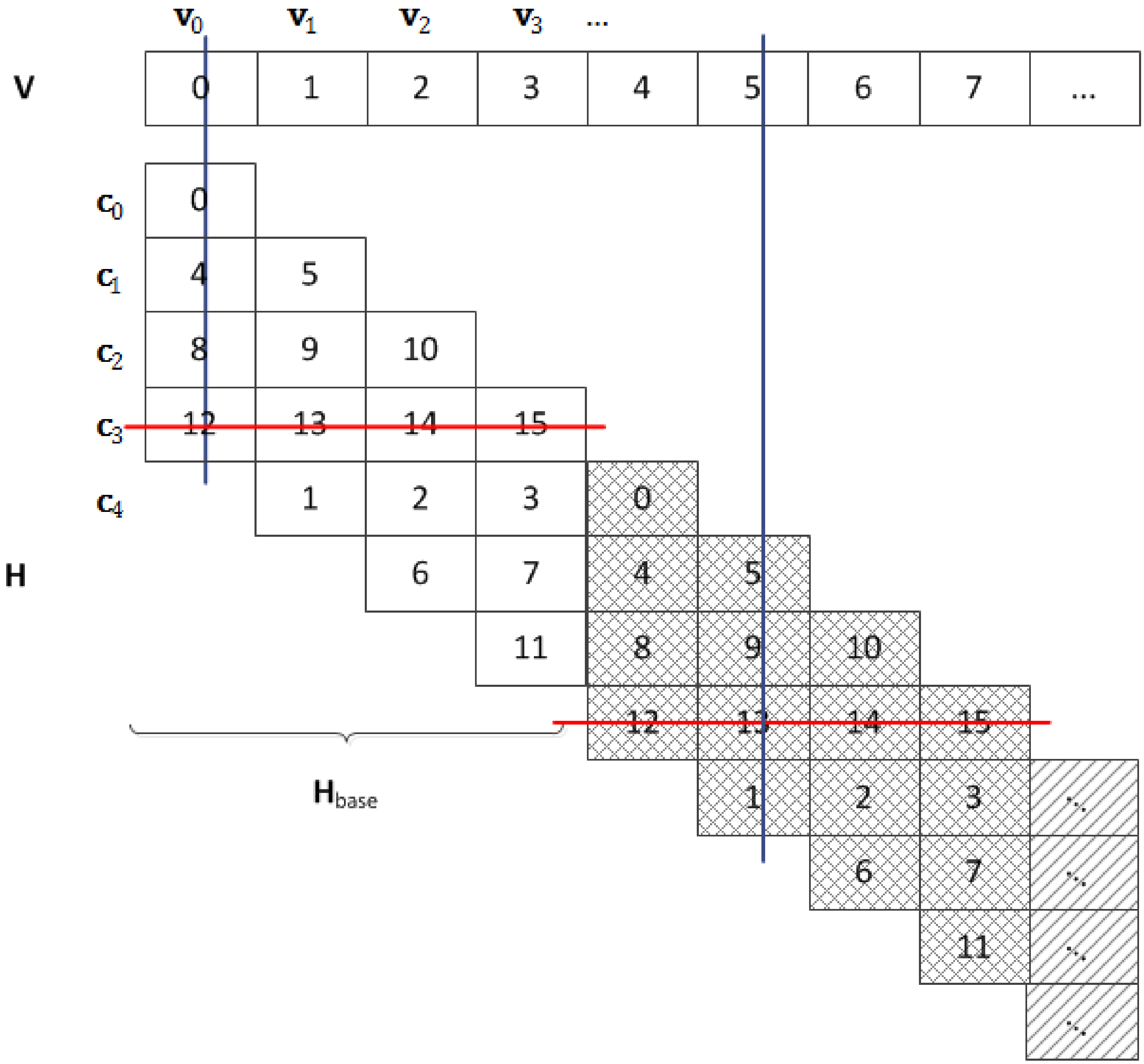
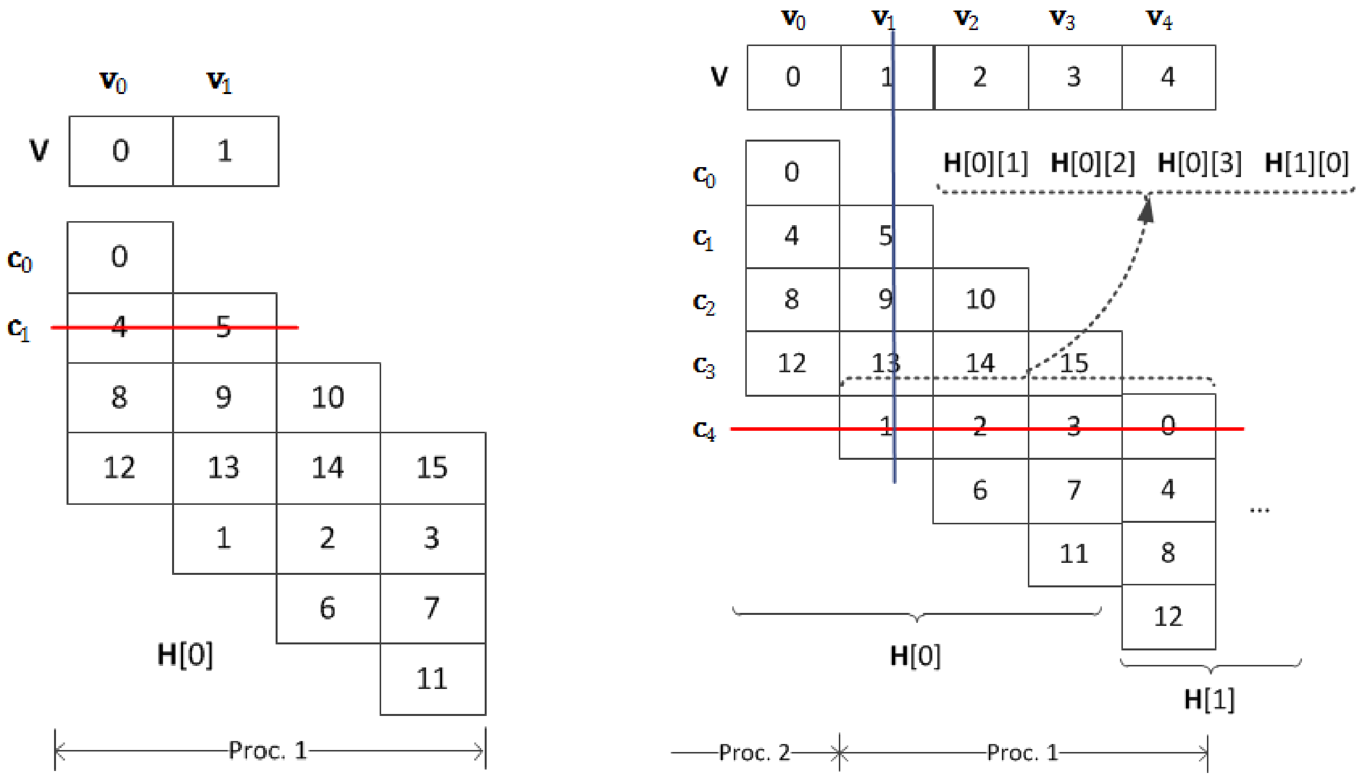


Fig. 7: The periodic structure of the parity-check matrix of the LDPCCCs.



(a) Updating at time slot 1.

(b) Updating at time slot 4.

Fig. 8: Illustration of the procedures of a LPDCCC decoder. The horizontal line denotes the updating of the row. The vertical line denotes the updating of a column.

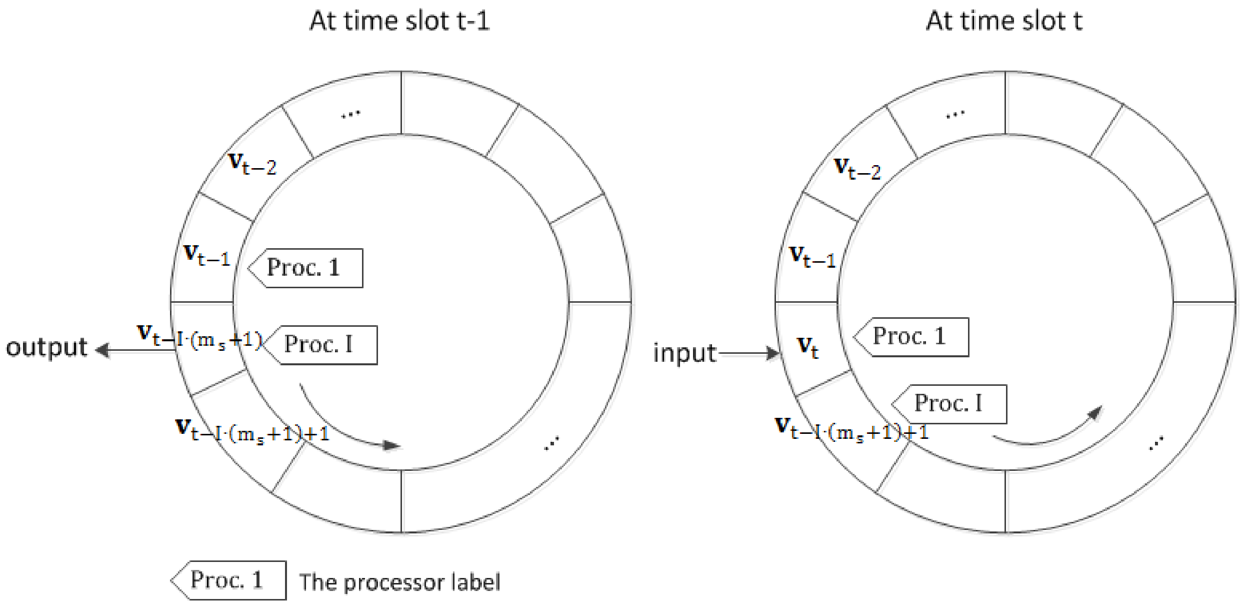


Fig. 9: Illustration of the circulant structure of LPDCCC decoder.